

# Fully-online construction of suffix trees and DAWGs for multiple texts

Takuya Takagi<sup>1</sup> Shunsuke Inenaga<sup>2</sup> Hiroki Arimura<sup>1</sup>

<sup>1</sup> Graduate School of IST, Hokkaido University, Japan  
 {tkg,arim}@ist.hokudai.ac.jp

<sup>2</sup> Department of Informatics, Kyushu University, Japan  
 inenaga@inf.kyushu-u.ac.jp

## Abstract

We consider *fully-online* construction of indexing data structures for multiple texts. Let  $\mathcal{T} = \{T_1, \dots, T_K\}$  be a collection of texts. By fully-online, we mean that a new character can be appended to any text in  $\mathcal{T}$  at any time. This is a natural generalization of *semi-online* construction of indexing data structures for multiple texts in which, after a new character is appended to the  $k$ th text  $T_k$ , then its previous texts  $T_1, \dots, T_{k-1}$  will remain static. Our fully-online scenario arises when we index multi-sensor data. We propose fully-online algorithms which construct the *directed acyclic word graph* (DAWG) and the *generalized suffix tree* (GST) for  $\mathcal{T}$  in  $O(N \log \sigma)$  time and  $O(N)$  space, where  $N$  and  $\sigma$  denote the total length of texts in  $\mathcal{T}$  and the alphabet size, respectively.

## 1 introduction

Text indexing is a fundamental problem in computer science, which plays important roles in many applications including text retrieval, molecular biology, signal processing, and sensor data analysis. In this paper, we focus on indexing a collection of multiple texts, so that subsequent pattern matching queries can be answered quickly. In particular, we study online indexing for a collection  $\mathcal{T}$  of multiple texts, where a new character can be appended to each text at any time. Such fully-online indexing for multiple growing texts has potential applications to continuous processing of data streams, where a number of symbolic events or data items are produced from multiple, rapid, time-varying, and unbounded data streams [2, 10]. For example, motif mining system tries to discover characteristic or interesting collective behaviors, such as frequent path or anomalies, from data streams generated by a collection of moving objects or sensors [10, 12].

It is known that suffix trees [11] and DAWGs [3] can be constructed for a collection of growing texts in the *semi-online* setting, where only the last inserted text can be grown. However, these existing semi-online algorithms to maintain a suffix tree or a DAWG for multiple texts are not sufficient to construct indexing structures for multiple data streams which grow in a fully-online manner.

We propose how the DAWG and the suffix tree can be incrementally constructed for a fully-online text collection. First, we observe that Blumer et al.'s construction [3] for DAWGs and Weiner's right-to-left construction [13] for suffix trees can readily be adapted to solve this problem. Hence, at any moment during the fully-online growth of the texts, we can find all *occ* occurrences of a given pattern of length  $M$  in the current text collection in  $O(M \log \sigma + \text{occ})$  time.

Our next goal is to extend Ukkonen's construction [11] to fully-online left-to-right construction of suffix trees for multiple texts. A motivation of this goal is that a growing suffix tree can be enhanced with powerful semi-dynamic tree data structures such as those for *nearest marked ancestor (NMA) queries* [14], *lowest common ancestor (LCA) queries* [7], and *level ancestor (LA) queries* [1]. Note that these data structures cannot be applied to DAWGs, and that the same query results cannot be obtained on the suffix tree maintained in a Weiner-like right-to-left manner since the suffix tree obtained in this manner inherently indexes the *reversed* texts in the collection. However, it turns out that this goal is a big algorithmic challenge, because: (A) In Ukkonen's algorithm, a pointer called the *active point* keeps track of the insertion points of suffixes in decreasing order of length. The efficiency of Ukkonen's algorithm is due to the monotonicity of the tracking path of the active point. However, unfortunately this monotonicity does not hold in our fully-online construction for multiple texts. (B) Due to the non-monotonicity mentioned above, Ukkonen's technique to amortize the cost to track the suffix insertion points does not work in our

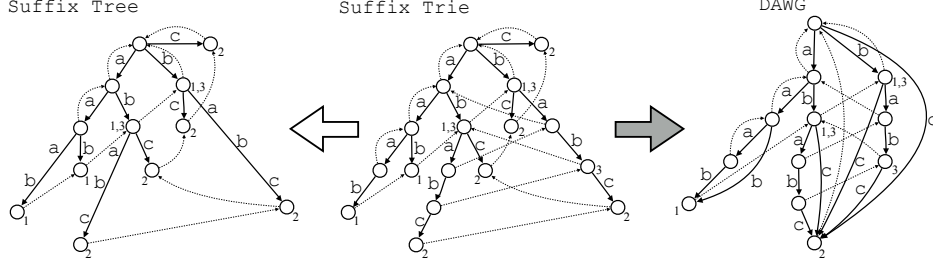


Figure 1: Illustration for  $STrie(\mathcal{T})$ ,  $STree(\mathcal{T})$ , and  $DAWG(\mathcal{T})$  with  $\mathcal{T} = \{T_1 = \text{aaab}, T_2 = \text{ababc}, T_3 = \text{bab}\}$ . The solid arrows and broken arrows represent the edges and the suffix links of each data structure, respectively. The number  $k$  ( $k = 1, 2, 3$ ) beside each node indicates that the node represents a suffix of  $T_k$ . The nodes  $[\text{ab}]_{\mathcal{T}}$  and  $[\text{b}]_{\mathcal{T}}$  are separated in  $DAWG(\mathcal{T})$  since the node  $\text{bab}$  in  $STrie(\mathcal{T})$  represents a suffix of  $T_3$ , while the node  $\text{abab}$  does not (see also the subtrees rooted at nodes  $\text{ab}$  and  $\text{b}$  in  $STrie(\mathcal{T})$ ).

case. (C) Ukkonen’s “open edge” technique to maintain the leaves does not work in our case, either. In Section 5 we will explain in more details why and how these problems arise in our fully-online setting. In this paper, we present a number of new novel techniques to overcome all the difficulties above. As a final result, we propose the first *optimal*  $O(N \log \sigma)$ -time  $O(N)$ -space fully-online left-to-right construction algorithm for a suffix tree of multiple texts over a general ordered alphabet of size  $\sigma$ , where  $N$  is the final total length of the texts.

**Related work:** We note that we can obtain fully-online text index for multiple texts using existing more general dynamic text indices as follows. For the index of Ferragina and Grossi [8] which permits character-wise updates, first we build a master text  $\$1 \cdots \$K$  consisting of  $K$  delimiters. Then, appending a character  $a$  to the  $k$ th text in the collection reduces to prepending  $a$  to the  $k$ th delimiter  $\$k$ . Using this approach, the index of Ferragina and Grossi [8] takes  $O(N \log N)$  total time to be constructed, requires  $O(N \log N)$  space, and allows pattern matching in  $O(M + \log N + N \log M + occ)$  time. For the compressed index for a dynamic text collection of Chan et al. [6], we can append a new character  $a$  to the  $k$ th text  $T_k$  by removing  $T_k$  and then adding  $T_k a$  in  $O(|T_k|)$  time. This yields a fully-online index with  $O(N^2 \log N)$  construction time and  $O(N)$  bits of space (or  $O(N / \log N)$  words of space assuming  $\Theta(\log N)$ -bit machine word), supporting pattern matching in  $O(M \log N + occ \log^2 N)$  time.

## 2 Preliminaries

**Strings:** Let  $\Sigma$  be a general ordered alphabet. Any element of  $\Sigma^*$  is called a *string*. For any string  $T$ , let  $|T|$  denote its length. Let  $\varepsilon$  be the empty string, namely,  $|\varepsilon| = 0$ . If  $T = XYZ$ , then  $X$ ,  $Y$ , and  $Z$  are called a *prefix*, a *substring*, and a *suffix* of  $T$ , respectively. For any  $1 \leq i \leq j \leq |T|$ , let  $T[i..j]$  denote the substring of  $T$  that begins at position  $i$  and ends at position  $j$  in  $T$ . For any  $1 \leq i \leq |T|$ , let  $T[i]$  denote the  $i$ th character of  $T$ . For any string  $T$ , let  $\text{Suffix}(T)$  denote the set of suffixes of  $T$ , and for any set  $\mathcal{T}$  of strings, let  $\text{Suffix}(\mathcal{T})$  denote the set of suffixes of all strings in  $\mathcal{T}$ . Namely,  $\text{Suffix}(\mathcal{T}) = \bigcup_{T \in \mathcal{T}} \text{Suffix}(T)$ . For any string  $T$ , let  $\bar{T}$  denote the reversed string of  $T$ , i.e.,  $\bar{T} = T[|T|] \cdots T[1]$ .

Let  $\mathcal{T} = \{T_1, \dots, T_K\}$  be a collection of  $K$  texts. For any  $1 \leq k \leq K$ , let  $\text{lrs}_{\mathcal{T}}(T_k)$  be the longest suffix of  $T_k$  that occurs at least twice in  $\mathcal{T}$ .

**Suffix trees and DAWGs for multiple texts:** The suffix trie for a text collection  $\mathcal{T} = \{T_1, \dots, T_K\}$ , denoted  $STrie(\mathcal{T})$ , is a trie which represents  $\text{Suffix}(\mathcal{T})$ . The size of  $STrie(\mathcal{T})$  is  $O(N^2)$ , where  $N$  is the total length of texts in  $\mathcal{T}$ . We identify each node  $v$  of  $STrie(\mathcal{T})$  with the string that  $v$  represents. A substring  $x$  of a text in  $\mathcal{T}$  is said to be *branching* in  $\mathcal{T}$ , if there exist two distinct characters  $a, b \in \Sigma$  such that both  $xa$  and  $xb$  are substrings of some texts in  $\mathcal{T}$ . Clearly, node  $x$  of  $STrie(\mathcal{T})$  is branching iff  $x$  is branching in  $\mathcal{T}$ . For each node  $av$  of  $STrie(\mathcal{T})$  with  $a \in \Sigma$  and  $v \in \Sigma^*$ , let  $\text{slink}(av) = v$ . This auxiliary edge  $\text{slink}(av) = v$  from  $av$  to  $v$  is called a *suffix link*.

The *suffix tree* [13] for a text collection  $\mathcal{T}$ , denoted  $STree(\mathcal{T})$ , is a “compacted trie” which represents  $\text{Suffix}(\mathcal{T})$ .  $STree(\mathcal{T})$  is obtained by compacting every path of  $STrie(\mathcal{T})$  which consists of non-branching internal nodes (see Fig. 1). Since every internal node of  $STree(\mathcal{T})$  is branching, and since there are at most  $N$  leaves in  $STree(\mathcal{T})$ , the numbers of edges and nodes are  $O(N)$ . The edge labels of  $STree(\mathcal{T})$  are non-empty substrings of some text in  $\mathcal{T}$ . By representing each edge label  $x$  with a triple  $\langle k, i, j \rangle$  of integers s.t.  $x = T_k[i..j]$ ,  $STree(\mathcal{T})$  can be stored with  $O(N)$  space. We say that any branching (resp. non-branching) substring of  $\mathcal{T}$  is an *explicit node* (resp. *implicit node*) of  $STree(\mathcal{T})$ . An implicit node  $x$

is represented by a triple  $(v, a, \ell)$ , called a *reference* to  $x$ , such that  $v$  is an explicit ancestor of  $x$ ,  $a$  is the first character of the path from  $v$  to  $x$ , and  $\ell$  is the length of the path from  $v$  to  $x$ . A reference  $(v, a, \ell)$  to node  $x$  is called *canonical* if  $v$  is the lowest explicit ancestor of  $x$ . For each node  $av$  of  $S\text{Tree}(\mathcal{T})$  with  $a \in \Sigma$  and  $v \in \Sigma^*$ , let  $\text{slink}(av) = v$ .

The *directed acyclic word graph* [3, 4] of a text collection  $\mathcal{T}$ , denoted  $\text{DAWG}(\mathcal{T})$ , is a smallest DAG which represents  $\text{Suffix}(\mathcal{T})$ .  $\text{DAWG}(\mathcal{T})$  is obtained by merging identical subtrees of  $S\text{Trie}(\mathcal{T})$  connected by the suffix links (see Fig. 1). Hence, the label of every edge of  $\text{DAWG}(\mathcal{T})$  is a single character. The numbers of nodes and edges of  $\text{DAWG}(\mathcal{T})$  are  $O(N)$  [3], and hence  $\text{DAWG}(\mathcal{T})$  can be stored with  $O(N)$  space.  $\text{DAWG}(\mathcal{T})$  can be defined formally as follows: For any string  $x$ , let  $\text{Epos}_{\mathcal{T}}(x)$  be the set of ending positions of  $x$  in the texts in  $\mathcal{T}$ , i.e.,  $\text{Epos}_{\mathcal{T}}(x) = \{(k, j) \mid x = T_k[j - |x| + 1..j], 1 \leq j \leq |T_k|, 1 \leq k \leq K\}$ . Consider an equivalence relation  $\equiv_{\mathcal{T}}$  on substrings  $x, y$  of texts in  $\mathcal{T}$  such that  $x \equiv_{\mathcal{T}} y$  iff  $\text{Epos}_{\mathcal{T}}(x) = \text{Epos}_{\mathcal{T}}(y)$ . For any substring  $x$  of texts of  $\mathcal{T}$ , let  $[x]_{\mathcal{T}}$  denote the equivalence class w.r.t.  $\equiv_{\mathcal{T}}$ . There is a one-to-one correspondence between each node  $v$  of  $\text{DAWG}(\mathcal{T})$  and each equivalence class  $[x]_{\mathcal{T}}$ , and hence we will identify each node  $v$  of  $\text{DAWG}(\mathcal{T})$  with its corresponding equivalence class  $[x]_{\mathcal{T}}$ . Let  $\text{long}([x]_{\mathcal{T}})$  denote the longest member of  $[x]_{\mathcal{T}}$ . By the definition of equivalence classes,  $\text{long}([x]_{\mathcal{T}})$  is unique for each  $[x]_{\mathcal{T}}$  and every member of  $[x]_{\mathcal{T}}$  is a suffix of  $\text{long}([x]_{\mathcal{T}})$ . If  $x, xa$  are substrings of texts in  $\mathcal{T}$  with  $x \in \Sigma^*$  and  $a \in \Sigma$ , then there exists an edge labeled with character  $a \in \Sigma$  from node  $[x]_{\mathcal{T}}$  to node  $[xa]_{\mathcal{T}}$ . This edge is called *primary* if  $|\text{long}([x]_{\mathcal{T}})| + 1 = |\text{long}([xa]_{\mathcal{T}})|$ , and is called *secondary* otherwise. For each node  $[x]_{\mathcal{T}}$  of  $\text{DAWG}(\mathcal{T})$  with  $|x| \geq 1$ , let  $\text{slink}([x]_{\mathcal{T}}) = y$ , where  $y$  is the longest suffix of  $\text{long}([x]_{\mathcal{T}})$  which does not belong to  $[x]_{\mathcal{T}}$ . In the example of Fig. 1,  $[\text{aaab}]_{\mathcal{T}} = \{\text{aaab}, \text{aab}\}$ . The edge labeled with **b** from node  $[\text{aaa}]_{\mathcal{T}}$  to node  $[\text{aaab}]_{\mathcal{T}}$  is primary, while the edge labeled with **b** from node  $[\text{aa}]_{\mathcal{T}}$  to node  $[\text{aaab}]_{\mathcal{T}}$  is secondary.  $\text{slink}([\text{aaab}]_{\mathcal{T}}) = [\text{ab}]_{\mathcal{T}}$ .

The following fact follows from the definition of branching substrings:

**Fact 1.** *For any substring  $x$  of texts in  $\mathcal{T}$ , node  $x$  is branching (explicit) in  $S\text{Tree}(\mathcal{T})$  iff node  $[x]_{\mathcal{T}}$  is branching in  $\text{DAWG}(\mathcal{T})$ .*

**Fully-online text collection:** We consider a collection  $\{T_1, \dots, T_K\}$  of  $K$  growing texts, where each text  $T_k$  ( $1 \leq k \leq K$ ) is initially the empty string  $\varepsilon$ . Given a pair  $(k, a)$  of a text id  $k$  and a character  $a \in \Sigma$  which we call an *update operator*, the character  $a$  is appended to the  $k$ -th text of the collection. For a sequence  $U$  of update operators, let  $U[1..i]$  denote the sequence of the first  $i$  update operators in  $U$  with  $0 \leq i \leq |U|$ . Also, for  $0 \leq i \leq |U|$  let  $\mathcal{T}_{U[1..i]}$  denote the collection of texts which have been updated according to the first  $i$  update operators of  $U$ . For instance, consider a text collection of three texts which grow according to the following sequence  $U = (1, \text{a}), (2, \text{b}), (2, \text{a}), (3, \text{a}), (1, \text{a}), (3, \text{c}), (3, \text{b}), (2, \text{b}), (1, \text{a}), (1, \text{b}), (3, \text{c}), (3, \text{b}), (1, \text{c}), (3, \text{b}), (2, \text{c})$  of 15 update operators. Then,

$$\mathcal{T}_{U[1..0]} = \left\{ \begin{array}{c} \varepsilon \\ \varepsilon \\ \varepsilon \end{array} \right\}, \dots, \mathcal{T}_{U[1..14]} = \left\{ \begin{array}{ccccc} 1 & 5 & 9 & 10 & 13 \\ \text{a} & \text{a} & \text{a} & \text{b} & \text{c} \\ 2 & 3 & 8 & & \\ \text{b} & \text{a} & \text{b} & & \\ 4 & 6 & 7 & 11 & 12 & 14 \\ \text{a} & \text{c} & \text{b} & \text{c} & \text{b} & \text{b} \end{array} \right\}, \mathcal{T}_{U[1..15]} = \left\{ \begin{array}{ccccc} 1 & 5 & 9 & 10 & 13 \\ \text{a} & \text{a} & \text{a} & \text{b} & \text{c} \\ 2 & 3 & 8 & 15 & \\ \text{b} & \text{a} & \text{b} & \text{c} & \\ 4 & 6 & 7 & 11 & 12 & 14 \\ \text{a} & \text{c} & \text{b} & \text{c} & \text{b} & \text{b} \end{array} \right\}$$

where the superscript  $i$  over each character  $a$  in the  $k$ -th text implies that  $U[i] = (k, a)$ . For instance,  $U[15] = (2, \text{c})$  and hence **c** was appended to the 2nd text  $T_2 = \text{bab}$  in  $\mathcal{T}_{U[1..14]}$ , yielding  $T_2 = \text{babc}$  in  $\mathcal{T}_{U[1..15]}$ .

If there is no restriction on  $U$  like the one in the example above, then  $U$  is called *fully-online*. If there is a restriction on  $U$  such that once a new character is appended to the  $k$ -th text, then no characters will be appended to its previous  $k - 1$  texts, then  $U$  is called *semi-online*. Hence, any semi-online sequence of update operators is of form  $(1, T_1[1]), \dots, (1, T_1[|T_1|]), \dots, (K, T_K[1]), \dots, (K, T_K[|T_K|])$ .

Section 3 reviews previous algorithms which incrementally construct the DAWG and the suffix tree for a growing text collection in the semi-online setting. Sections 4 and 5 propose our new algorithms which incrementally construct the DAWG and the suffix tree for a text collection in the fully-online setting, respectively.

### 3 Semi-online construction algorithms

**Blumer et al.'s semi-online DAWG construction algorithm:** We recall Blumer et al.'s algorithm [3] which incrementally builds  $\text{DAWG}(\mathcal{T}_U)$  for a given semi-online sequence  $U$  of update operators of length  $N$ . Since  $U$  is semi-online, at each step  $i$  ( $0 \leq i \leq N$ ) of the semi-online update, there exists a unique  $k$  ( $1 \leq k < K$ ) such that  $T_1, \dots, T_{k-1}$  will be static for all the following  $i'$ th steps ( $i \leq i' \leq N$ ),  $T_k$  is

now growing from left to right, and  $T_{k+1}, \dots, T_K$  are still the empty strings. Assume that  $U[i] = (k, a)$ , and hence a new character  $a$  is appended to the  $k$ th text in the collection at the  $i$ th step. For ease of notation, let  $\mathcal{T}' = \mathcal{T}_{U[1..i-1]}$  and  $\mathcal{T} = \mathcal{T}_{U[1..i]}$ . Also, assume that  $DAWG(\mathcal{T}')$  has already been constructed. In updating  $DAWG(\mathcal{T}')$  to  $DAWG(\mathcal{T})$ , we have to assure that all suffixes of the extended text  $T_k a$  will be represented by  $DAWG(\mathcal{T})$ . These suffixes are categorized to three different types (see also Fig. 2 in Appendix A):

**Type-1** The suffixes of  $T_k a$  that are longer than  $lrs_{\mathcal{T}'}(T_k)a$ .

**Type-2** The suffixes of  $T_k a$  that are not longer than  $lrs_{\mathcal{T}'}(T_k)a$  and are longer than  $lrs_{\mathcal{T}}(T_k a)$ .

**Type-3** The suffixes of  $T_k a$  that are not longer than  $lrs_{\mathcal{T}}(T_k a)$ .

Blumer et al's algorithm inserts the suffixes of  $T_k a$  in decreasing order of length, from the Type-1 ones to the Type-2 ones. By definition, the Type-3 ones are already represented by  $DAWG(\mathcal{T}')$ , and hence, we need not insert them explicitly.

Their algorithm maintains an invariant  $v$  which indicates node  $[T_k]_{\mathcal{T}'}$ , called the *active point*, from which the update starts. There are two cases to happen:

1. If there is an out-going edge labeled with  $a$  from  $v$ , then  $T_k a = lrs_{\mathcal{T}}(T_k a)$ , which implies all suffixes of  $T_k a$  are of Type-3. There are two subcases:
  - (a) If the edge labeled with  $a$  is primary, then no updates to the graph topology are needed. The new active point for the next step is on  $[lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}}$ .
  - (b) If the edge labeled with  $a$  is secondary, then the graph topology needs to be updated (see Fig. 3 in Appendix A). Since the edge is secondary, every member  $Xa$  of  $u = [lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}'}$  that is longer than  $T_k a$  is not a suffix of  $T_k a$ , while every member  $Ya$  of  $u = [lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}'}$  that is not longer than  $T_k a$  is a Type-3 suffix of  $T_k a$ . This implies that  $Epos_{\mathcal{T}}(lrs_{\mathcal{T}}(T_k a)) \supset Epos_{\mathcal{T}}(Xa)$ . By the definition of the nodes of DAWGs (recall Subsection 2), the node  $u$  is split into two nodes  $z = [Xa]_{\mathcal{T}}$  and  $w = [lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}}$ : First, a new node  $w$  is created. All secondary incoming edges of  $u$  corresponding to Type-3 suffixes  $Ya$  are redirected to  $w$ . This can be done by traversing the chain of the suffix links starting from  $v$ . All the out-going edges of  $u$  are copied to  $w$ . Now, node  $w$  is complete, and the node  $u$  with its remaining in-coming edges is the other new node  $z$ . The suffix link of  $u$  is inherited by  $w$ , and the suffix link of  $z$  is set to  $w$ . The new active point for the next step is on node  $w$ .
2. If there is no out-going edge labeled with  $a$  from the active point  $v$ , then a new sink  $s$  is created. The Type-1 suffixes are inserted by making a new edge labeled by  $a$  from  $v = [T_k]_{\mathcal{T}'}$  to  $s$ . To insert the Type-2 suffixes, the active point  $v$  moves by updating  $v \leftarrow slink(v)$ . Then the following procedure is repeated until an out-going edge labeled with  $a$  from the active point is found: (i) A new edge labeled with  $a$  from  $v$  to  $s$  is created. (ii) The active point  $v$  moves by updating  $v \leftarrow slink(v)$ . The node  $u$  where the above procedure ends is  $[lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}'}$ , and the new sink  $s$  is exactly  $[T_k a]_{\mathcal{T}}$  which represent all Type-1 and Type-2 suffixes of  $T_k a$ . There are two cases:
  - (a) If the edge labeled with  $a$  from the last locus  $v$  of the active point to  $u$  is primary, then  $u = [lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}}$ . Thus no updates to the graph topology are needed. The suffix link of the new sink  $s = [T_k a]_{\mathcal{T}}$  is set to  $u$ .
  - (b) If the edge labeled with  $a$  from the last locus  $v$  of the active point to  $u$  is secondary, then as in Case 1b,  $u$  is split into two nodes  $w$  and  $z$  where  $w$  represents the members of  $u$  that are longer than the longest repeating suffix  $lrs_{\mathcal{T}}(T_k a)$  (none of these members is a suffix of  $T_k a$ ), and  $z$  represents the members of  $u$  which are Type-3 suffixes of  $T_k a$ . The suffix link of the new sink  $s$  is set to  $z$ .

In both subcases above, the new active point is on the new sink  $s = [T_k a]_{\mathcal{T}}$ .

It is not difficult to see that if the total number of new nodes, edges, and suffix links is  $q$ , then the above update takes  $O(q \log \sigma)$  time, where the  $\log \sigma$  term is due to searching for an out-going edge labeled by  $a$ . Since no existing nodes, edges, or suffix links are deleted during the updates, and since the size of  $DAWG(\mathcal{T}_U)$  is  $O(N)$ , the amortized time for the update is  $O(\log \sigma)$ . Hence,  $DAWG(\mathcal{T}_U)$  can be constructed in  $O(N \log \sigma)$  time and  $O(N)$  space in the semi-online setting.

**Ukkonen's semi-online suffix tree construction algorithm:** Ukkonen [11] proposed an algorithm to incrementally construct the suffix tree of a single text. His algorithm can easily be extended to incrementally construct the suffix tree for multiple texts in the semi-online setting.

Let  $U$  be a semi-online sequence of  $N$  update operators such that the last update operator for each  $k$  ( $1 \leq k \leq K$ ) is  $(k, \$_k)$ , where  $\$_k$  is a special end-marker for the  $k$ th text in the collection. For ease of notation,  $\mathcal{T}' = \mathcal{T}_{u[1..i-1]}$  and  $\mathcal{T} = \mathcal{T}_{u[1..i]}$ . Also, assume that we have already constructed  $STree(\mathcal{T}')$  and that the next update operator is  $U[i] = (k, a)$ . Thus a new character  $a$  is appended to the  $k$ th text  $T_k$  of  $\mathcal{T}'$ , and the  $k$ th text of  $\mathcal{T}$  becomes  $T_k a$ .

As in the case of semi-online DAWG construction, the suffixes of  $T_k a$  are inserted in decreasing order of length. The Type-1 suffixes are maintained as follows. Let  $s$  be any suffix of  $T_k$  which is represented by a leaf of  $STree(\mathcal{T}')$ . Since  $s$  is a non-repeating suffix of  $T_k$  in  $\mathcal{T}'$ ,  $sa$  is a non-repeating suffix of  $T_k a$  in  $\mathcal{T}$ , which implies that  $sa$  will also be a leaf of  $STree(\mathcal{T})$ . Based on this observation, the label of the in-coming edge of  $s$  is represented by a triple  $\langle k, b, \infty \rangle$  called an *open edge*, where  $b$  is the beginning position of the label of the in-coming edge in the  $k$ th text. This way, every existing leaf will then be automatically extended. Hence, updating  $STree(\mathcal{T}')$  to  $STree(\mathcal{T})$  reduces to inserting the Type-2 suffixes of  $T_k a$ . For this sake, the algorithm maintains an invariant which indicates the locus of  $x = lrs_{\mathcal{T}'}(T_k)$  on  $STree(\mathcal{T}')$  called the *active point*. Since  $x$  can be an implicit node, the algorithm maintains the canonical reference  $(v, c, \ell)$  to  $x$ . For convenience, if  $x$  is an explicit node, then let its canonical reference be  $(x, \varepsilon, 0)$ . The update starts from the current active point  $x$  represented by its canonical reference pair, and the Type-2 suffixes of  $T_k a$  are inserted in decreasing order of length, by using the chain of (virtual) suffix links. There are two cases:

- I. If it is possible to go down from  $x$  with character  $a$ , then no updates to the tree topology are needed. The new active point is  $xa$ , and the reference to  $xa$  is made canonical if necessary. The update ends.
- II. If it is impossible to go down from  $x$  with character  $a$ , then we create a new leaf. Let  $j$  be the beginning position of the suffix of  $T_k a$  which corresponds to this new leaf. The following procedure is repeated until Case I happens.
  - (a) If the active point  $x$  is on an explicit node, then a new leaf node  $s$  is created as a new child of  $x$ , with its incoming edge labeled by  $\langle k, b, \infty \rangle$ , where  $b = |T_k a| - |x| + 1$ . The active point  $x$  is updated to  $slink(x)$ .
  - (b) If the active point  $x$  is on an implicit node, then  $x$  becomes explicit in this step. A new leaf node  $s$  is created as a new child of  $x$  with its incoming edge labeled by  $\langle k, b, \infty \rangle$ . Since the suffix link of the new explicit node  $x$  does not yet exist, we simulate the suffix link traversal as follows (see also Fig. 4 in Appendix A). Let  $(v_j, c_j, \ell_j)$  be the canonical reference to  $x$ . First, we follow the suffix link  $slink(v_j)$  of  $v_j$ , and then go down along the path of length  $\ell_j$  from  $slink(v_j)$  starting with character  $c_j$ . Let this locus be  $x'$ . Let  $v_{j+1}$  be the longest explicit node in this path. (i) If  $|v_{j+1}| = |x'|$ , then we firstly create the new suffix link  $slink(x) = v_{j+1}$  for the new explicit node  $x$ . The active point  $x$  is updated to  $x'$  and is represented by canonical reference  $(v_{j+1}, \varepsilon, 0)$ . (ii) If  $|v_{j+1}| < |x'|$ , then the next active point is implicit. The active point  $x$  is updated to  $x'$  and is represented by canonical reference  $(v_{j+1}, c_{j+1}, \ell_{j+1})$ . The suffix link of  $x$  will be set to  $x'$  when  $x'$  becomes explicit in the next step.

The most expensive case is II-b-(ii). Since the path from  $v_{j+1}$  to  $x'$  contains at most  $\ell_j - \ell_{j+1}$  explicit nodes, it takes  $O((\ell_j - \ell_{j+1} + 1) \log \sigma)$  time to locate the next active point  $x'$  (note  $\ell_j - \ell_{j+1} \geq 0$  holds). All the other operations take  $O(\log \sigma)$  time. Hence, the total cost to insert all leaves (suffixes) for the  $k$ th text is  $O(\sum_{j=1}^{N_k} (\ell_j - \ell_{j+1} + 1) \log \sigma) = O(N_k \log \sigma)$ , where  $N_k$  is the final length of the  $k$ th text. Thus the amortized time cost for each leaf (suffix) for the  $k$ th text is  $O(\log \sigma)$ . Overall, it takes a total of  $O(N \log \sigma)$  time to construct  $STree(\mathcal{T}_U)$  for a semi-online sequence  $U$  of update operators. The space requirement is  $O(N)$ .

## 4 Fully-online DAWG construction algorithm

We can easily extend Blumer et al.'s semi-online DAWG construction algorithm to the fully-online setting. Let  $U$  be a fully-online sequence of  $N$  update operators. Our fully-online algorithm maintains the active point  $v_k$  for *every* growing text  $T_k$  in the collection, at any step of the algorithm. Now, assume that we have already constructed  $DAWG(\mathcal{T}')$ , where  $\mathcal{T}' = \mathcal{T}_{U[1..i-1]}$  for  $1 \leq i \leq N$ . Let  $U[i] = (k, a)$ , and we are updating  $DAWG(\mathcal{T}')$  to  $DAWG(\mathcal{T})$ , where  $\mathcal{T} = \mathcal{T}_{U[1..i]}$ . The update starts from the active point  $v_k = [T_k]_{\mathcal{T}'}$ , exactly in the same way as was described in Section 3. The total cost to update  $DAWG(\mathcal{T}')$  to  $DAWG(\mathcal{T})$  is again  $O(q \log \sigma)$ , where  $q$  is the total number of nodes, edges, and suffix links which were introduced in this update. Since the total size of  $DAWG(\mathcal{T})$  is  $O(N)$ , the amortized cost for this update is again  $O(\log \sigma)$ . By the above arguments, we obtain the following theorem.

**Theorem 1.** *Given a fully-online sequence  $U$  of  $N$  update operators for a collection of  $K$  texts, we can update  $DAWG(\mathcal{T}_{U[1..i]})$  for  $i = 1, \dots, N$  in a total of  $O(N \log \sigma)$  time and  $O(N)$  space.*

A snapshot of fully-online DAWG construction is shown in Fig. 5 of Appendix A.

Assume for now that each text  $T_k$  in a collection  $\mathcal{T}$  begins with a special character  $\#_k$  which does not appear elsewhere in  $\mathcal{T}$ . Then, the tree of the (reversed) suffix links of  $DAWG(\mathcal{T})$  forms the suffix tree  $S\text{Tree}(\overline{\mathcal{T}})$  for the collection  $\overline{\mathcal{T}} = \{\overline{T_1}, \dots, \overline{T_K}\}$  of the reversed texts of  $\mathcal{T}$  [3]. Hence, the next corollary follows from Theorem 1, which gives *right-to-left* fully-online suffix tree construction.

**Corollary 2.** *Given a fully-online sequence  $U$  of  $N$  update operators for a collection of  $K$  texts, we can update  $S\text{Tree}(\overline{\mathcal{T}_{U[1..i]}})$  for  $i = 1, \dots, N$  in a total of  $O(N \log \sigma)$  time and  $O(N)$  space.*

## 5 Fully-online suffix tree construction algorithm

**Difficulties in fully-online construction of suffix trees:** Unlike the case with DAWGs, it is not easy to extend Ukkonen’s semi-online suffix tree construction algorithm to our left-to-right fully-online setting, because:

- A. Let  $U[i] = (k, a)$  which updates the current  $k$ th text  $T_k$  to  $T_k a$ , and assume that we have just constructed  $S\text{Tree}(\mathcal{T}_{U[1..i]})$ . Recall that we defined the initial locus of the active point for  $T_k a$  on  $S\text{Tree}(\mathcal{T}_{U[1..i]})$  to be the longest repeating suffix of  $T_k a$  in  $\mathcal{T}_{U[1..i]}$ . However, since  $U$  is fully-online, any other text  $T_h$  ( $h \neq k$ ) in the collection would be updated by following update operators  $U[r]$  with  $r > i$ . Then, the longest repeating suffix of  $T_k a$  in  $\mathcal{T}_{U[1..r]}$  can be much longer than that of  $T_k a$  in  $\mathcal{T}_{U[1..i]}$ . In other words, some Type-1 suffixes of  $T_k a$  in  $\mathcal{T}_{U[1..i]}$  can become of Type-2 in  $\mathcal{T}_{U[1..r]}$  (see Fig. 6 in Appendix A for a concrete example). What is worse, updating  $T_h$  can affect the longest repeating suffix of any other text in the collection as well. If we maintain all these active points naïvely, it takes  $O(KN \log \sigma)$  time.
- B. Even if we somehow manage to efficiently maintain the active point for each text in the collection, there remains another difficulty. Let  $j$  be the beginning position of the longest repeating suffix of  $T_k a$  in  $\mathcal{T}_{U[1..i]}$ , and let  $(v_j, c_j, \ell_j)$  be the canonical reference to this suffix. Let  $U[i'] = (k, a')$  be the first update operator in  $U$  which updates the  $k$ th text after  $U[i] = (k, a)$ . Let  $(v'_j, c'_j, \ell'_j)$  be the canonical reference to the longest repeating suffix of  $T_k a$  in  $\mathcal{T}_{U[1..i']}$ , which is the “real” initial active point where insertion of the Type-2 suffixes should start at this  $i'$ th step. By the property of suffix trees  $\ell'_j \geq \ell_j$  holds, and what is worse, this length  $\ell'_j$  is unbounded by the number of Type-2 suffixes inserted at this  $i'$ th step. Thus, the amortization technique we used for the semi-online construction does not work in the fully-online setting.
- C. The phenomenon mentioned in Difficulty A also causes a problem of how to represent the labels of the in-coming edges to the leaves. Assume that we created a new leaf w.r.t. an update operator  $(k, a)$ , and let  $\langle k, b_k, \infty \rangle$  be the triple representing the label of the in-coming edge to the leaf, where  $b_k$  is the beginning position of the edge label in the  $k$ th text. It corresponds to a Type-1 suffix of the  $k$ th text, but the leaf can later be extended by another growing text  $T_h$ . Then, the triple  $\langle k, b_k, \infty \rangle$  has to be updated to  $\langle h, b_h, \infty \rangle$ , where  $b_h$  is the beginning position of the edge label in the  $h$ th text (see also Fig. 6 in Appendix A). Notice that this update may happen repeatedly.

**Constructing suffix trees with the aid of DAWGs:** We utilize DAWGs to overcome Difficulties A, B and C in fully-online construction of suffix trees. Namely, we construct  $S\text{Tree}(\mathcal{T})$  in tandem with  $DAWG(\mathcal{T})$ .

A high-level description of our algorithm is as follows. We insert the Type-2 suffixes of  $T_k a$  in *increasing order* of length, starting from the locus of the longest Type-3 suffix of  $T_k a$ . The idea of inserting the Type-2 suffixes in increasing order of length was also used by Breslauer and Italiano [5], for quasi real-time left-to-right construction of the suffix tree for a single text. To efficiently find the locus where the next longer Type-2 suffix should be inserted in the tree from the locus where the last Type-2 suffix was inserted, we introduce a simpler amortized variant of the *suffix tree oracle* of Fischer and Gawrychowski [9]. These will overcome Difficulties A and B. To overcome Difficulty C, we introduce new *lazy representation* of the labels of edges leading to the leaves.

**Lemma 1.** *We can compute, in amortized  $O(\log \sigma)$  time, a canonical reference to the longest Type-3 suffix  $lrs_{\mathcal{T}}(T_k a)$  of  $T_k a$  on  $S\text{Tree}(\mathcal{T}')$ , using a data structure which requires space linear in the total length of the texts in  $\mathcal{T}$ .*

*Proof.* We introduce the *longest path tree* of  $\mathcal{T}'$ , denoted  $LPT(\mathcal{T}')$ , which is the spanning tree of  $DAWG(\mathcal{T}')$  consisting only of the primary edges of  $DAWG(\mathcal{T}')$ . Every node of  $LPT(\mathcal{T}')$  is marked iff its corresponding node on  $DAWG(\mathcal{T}')$  is branching. Every marked node of  $LPT(\mathcal{T}')$  is linked to its corresponding node of  $S\mathcal{T}ree(\mathcal{T}')$  which is also branching by Fact 1 (see Fig. 7 in Appendix A).  $LPT(\mathcal{T}')$  is enhanced with the *nearest marked ancestor* (NMA) data structure of Westbrook [14], which supports the following operations in amortized  $O(1)$  time using linear space: 1) find the NMA of any node; 2) insert an unmarked node; 3) mark an unmarked node.

When  $DAWG(\mathcal{T}')$  is updated to  $DAWG(\mathcal{T})$ , at most two new primary edges are introduced to  $DAWG(\mathcal{T})$ , one for the new sink and one for the split node. We insert these new edges to  $LPT(\mathcal{T}')$  and obtain  $LPT(\mathcal{T})$ . Because of these new edges, at most two non-branching nodes of  $DAWG(\mathcal{T}')$  can become branching in  $DAWG(\mathcal{T})$ . We mark their corresponding nodes in  $LPT(\mathcal{T})$ , and link them to the corresponding suffix tree nodes after we have constructed  $S\mathcal{T}ree(\mathcal{T})$ . This is because the corresponding nodes of  $S\mathcal{T}ree(\mathcal{T}')$  are still non-branching.

We use  $LPT(\mathcal{T})$  to quickly move from the DAWG to the suffix tree. Since  $lrs_{\mathcal{T}}(T_k a)$  is the longest in  $[lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}}$ , there always exists a node  $y$  of  $LPT(\mathcal{T})$  which represents  $lrs_{\mathcal{T}}(T_k a)$ . We conduct an NMA query from  $y$  on  $LPT(\mathcal{T})$ , and let  $v$  be the NMA of  $y$ . Let  $\ell = |y| - |v|$ , and let  $c$  be the label of the first edge in the path from  $v$  to  $y$ . We move from  $v$  to its corresponding node  $x$  in  $S\mathcal{T}ree(\mathcal{T}')$ . Then,  $(x, c, \ell)$  is a reference to  $lrs_{\mathcal{T}}(T_k a)$  in  $S\mathcal{T}ree(\mathcal{T}')$ . Since  $v$  is the NMA of  $y$  in  $LPT(\mathcal{T})$ , and since updating  $T_k$  to  $T_k a$  does not explicitly insert any suffix shorter than  $lrs_{\mathcal{T}}(T_k a)$ , this reference is canonical by Fact 1.

Clearly the total size of the above data structures is linear in the total length of the texts in  $\mathcal{T}$ . We analyze the time complexity. Recall Case 2 when updating  $DAWG(\mathcal{T}')$  to  $DAWG(\mathcal{T})$ . At the end of the update, we find (or create) in amortized  $O(\log \sigma)$  time the node of  $DAWG(\mathcal{T})$  which represents  $[lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}}$ . Hence we can find node  $y = lrs_{\mathcal{T}}(T_k a)$  in amortized  $O(\log \sigma)$  time. Updating  $LPT(\mathcal{T}')$  to  $LPT(\mathcal{T})$  takes  $O(\log \sigma)$  time. Inserting a new node and querying an NMA from a given node takes amortized  $O(1)$  time. We can link a new marked node of  $LPT(\mathcal{T})$  to the corresponding new branching node of  $S\mathcal{T}ree(\mathcal{T})$  in  $O(1)$  time, since we can remember this new branching node when updating  $S\mathcal{T}ree(\mathcal{T}')$  to  $S\mathcal{T}ree(\mathcal{T})$ . Hence, the amortized bound is  $O(\log \sigma)$ .  $\square$

To find the insertion point of the shortest Type-2 suffix from the longest Type-3 suffix  $lrs_{\mathcal{T}}(T_k a)$ , and to insert the Type-2 suffixes of  $T_k a$  in increasing order of length, we maintain the labeled reversed suffix links for each explicit node of the suffix tree. Namely, if  $slink(bv) = v$  for two nodes  $bv, v$  with  $v \in \Sigma^*$  and  $b \in \Sigma$ , let  $rslink_b(v) = bv$ . We leave  $rslink_b(v)$  undefined if  $bv$  is not a substring of any text in the collection, or node  $bv$  is implicit in the suffix tree.

A *suffix tree oracle* for a suffix tree  $S$  is a data structure which efficiently answers the following query: given a pair  $(v, b)$  of a node of  $S$  and a character  $b \in \Sigma$ , return the nearest ancestor  $u$  of  $v$  for which  $rslink_b(u)$  is defined. The state-of-the-art suffix tree oracle by Fischer and Gawrychowski [9] answers queries and supports updates in worst-case  $O(\log \log n + (\log \log \sigma)^2 / \log \log \log \sigma)$  time each, using  $O(n)$  space, where  $n$  is the number of leaves in  $S$ . The next lemma shows our simpler suffix tree oracle with amortized  $O(\log \sigma)$  bound.

**Lemma 2.** *For a suffix tree with  $n$  leaves, there is a suffix tree oracle of size  $O(n)$  which answers each query in amortized  $O(\log \sigma)$  time. It takes amortized  $O(\log \sigma)$  time to update this suffix tree oracle, per insertion of a new leaf or a new suffix link to the suffix tree.*

*Proof.* (Sketch) We follow the approach by Fischer and Gawrychowski [9]. The  $\log \log n$  term in the running time of their suffix tree oracle is due to the fringe nearest marked ancestor data structure by Breslauer and Italiano [5], which answers each NMA query in a special case in worst case  $O(\log \log n)$  time. It is possible to replace the fringe nearest marked ancestor data structures with the NMA data structures of Westbrook [14], so the time cost for each NMA query is amortized to  $O(1)$ . The other  $(\log \log \sigma)^2 / \log \log \log \sigma$  term is due to fast predecessor data structures for integer alphabets. Since our alphabet is more general, we use balanced search trees with  $O(\log \sigma)$ -time operations. Hence our bound is  $O(\log \sigma)$  amortized. A complete proof is shown in Appendix B.  $\square$

To overcome Difficulty C, we employ lazy maintenance for leaves, namely, we maintain only the *first character* of the label of every edge leading to a leaf. On the other hand, we eagerly maintain the whole label of every edge leading to an internal explicit node. The next lemma holds.

**Lemma 3.** *The lazy representation of the in-coming edges of leaves allows for updating the suffix tree in amortized  $O(\log \sigma)$  time per insertion of a new leaf.*

*Proof.* Let  $U[i] = (k, a)$  and  $\mathcal{T} = \mathcal{T}_{U[1..i]}$  as previously. Let  $xa$  be a Type-2 suffix of the extended text  $T_k a$  to be inserted to the suffix tree. Using the suffix tree oracle of Lemma 2, we obtain a canonical reference  $(v, c, \ell)$  to  $x$  from which a leaf for the suffix  $xa$  is to be inserted.

The difficult case is when  $x$  is on the edge  $e$  from  $v$  to a leaf and  $\ell \geq 2$ , since we only know the first character  $c$  of the label of  $e$ . We create a new internal node  $x$  on  $e$ , and create a new leaf as a child of  $x$  and its in-coming edge labeled with the first character  $a$ . We can determine the label of the in-coming edge of the new internal explicit node  $x$  as follows. Let  $y$  be the node of  $LPT(\mathcal{T})$  which corresponds to the node  $[v]_{\mathcal{T}}$  of  $DAWG(\mathcal{T})$ , namely  $y = \text{long}([v]_{\mathcal{T}})$ . We represent the label of each edge of  $LPT(\mathcal{T})$  by a pair of the text id and the position of the character in the text of that id. Let  $\langle h, j \rangle$  be the label of the out-going edge of node  $y$  of  $LPT(\mathcal{T})$  such that  $T_h[j] = c$ . Since we insert the Type-2 suffixes of  $T_k a$  in increasing order of length, the path in  $LPT(\mathcal{T})$  of length  $\ell$  starting with this edge from  $y$  is non-branching. Thus, we can label the in-coming edge of the suffix tree by triple  $\langle h, j, j + \ell - 1 \rangle$ . See also Fig. 8 in Appendix A.

While updating  $DAWG(\mathcal{T}')$  to  $DAWG(\mathcal{T})$ , we have visited the node  $[x]_{\mathcal{T}}$ . We can obtain node  $y$  on  $LPT(\mathcal{T})$  by an NMA query from node  $\text{long}([x]_{\mathcal{T}})$ , and associate to  $y$  each Type-2 suffix  $xa$  of  $T_k a$  whose length is in range  $[s + 1, l + 1]$ , where  $s$  and  $l$  are the lengths of the shortest and longest members of  $[x]_{\mathcal{T}}$ , respectively. As we insert the Type-2 suffixes of  $T_k a$  to the suffix tree in increasing order of length, for each Type-2 suffix  $xa$  we can access to its corresponding node  $y$  in amortized  $O(\log \sigma)$  time. It takes amortized  $O(\log \sigma)$  time to query the suffix tree oracle by Lemma 2. All the other operations take  $O(1)$  time each.  $\square$

Assume we are searching a growing text collection  $\mathcal{T}$  for a given pattern  $P$ . If we stuck on the parent node  $u$  of a leaf in  $S\text{Tree}(\mathcal{T})$  due to our lazy leaf representation, then we can move to the  $DAWG$  node which corresponds to the parent node  $u$  via  $LPT(\mathcal{T})$ , and continue searching for  $P$  on  $DAWG(\mathcal{T})$ . This way we can find the locus of  $P$  on  $S\text{Tree}(\mathcal{T})$  in optimal  $O(M \log \sigma)$  time, where  $M = |P|$ . Also, since the tree topology is correctly maintained with our lazy leaf representation, semi-dynamic NMA [14], LCA [7], and LA [1] queries can be correctly supported in  $O(1)$  time on our suffix tree representation.

**Theorem 3.** *Given a fully-online sequence  $U$  of Nupdate operators for a collection of  $K$  texts, we can update  $S\text{Tree}(\mathcal{T}_{U[1..i]})$  for  $i = 1, \dots, N$  in a total of  $O(N \log \sigma)$  time and  $O(N)$  space.*

A snapshot of fully-online suffix tree construction is shown in Fig. 9 of Appendix A. After the whole  $U$  has been processed, we determine the triples representing the entire labels of the in-coming edges of all leaves of  $S\text{Tree}(\mathcal{T}_U)$  in a total of  $O(N)$  time. We can then discard  $DAWG(\mathcal{T}_U)$  and  $LPT(\mathcal{T}_U)$ .

## References

- [1] Alstrup, S., Holm, J.: Improved algorithms for finding level ancestors in dynamic trees. In: ICALP 2000. pp. 73–84 (2000)
- [2] Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. pp. 1–16. ACM (2002)
- [3] Blumer, A., Blumer, J., Haussler, D., McConnell, R., Ehrenfeucht, A.: Complete inverted files for efficient text retrieval and analysis. J. ACM 34(3), 578–595 (1987)
- [4] Blumer, A., Blumer, J., Haussler, D., Ehrenfeucht, A., Chen, M.T., Seiferas, J.: The smallest automaton recognizing the subwords of a text. TCS 40, 31–55 (1985)
- [5] Breslauer, D., Italiano, G.F.: Near real-time suffix tree construction via the fringe marked ancestor problem. J. Discrete Algorithms 18, 32–48 (2013)
- [6] Chan, H., Hon, W., Lam, T.W., the master text, K.S.: Compressed indexes for dynamic text collections. ACM Transactions on Algorithms 3(2) (2007)
- [7] Cole, R., Hariharan, R.: Dynamic LCA queries on trees. In: SODA 1999. pp. 235–244 (1999)
- [8] Ferragina, P., Grossi, R.: Improved dynamic text indexing. J. Algorithms 31(2), 291–319 (1999)
- [9] Fischer, J., Gawrychowski, P.: Alphabet-dependent string searching with wexponential search trees. In: CPM 2015. pp. 160–171 (2015), full version is available at <http://arxiv.org/abs/1302.3347>



- [10] Keogh, E.J., Lonardi, S., Chiu, B.Y.: Finding surprising patterns in a time series database in linear time and space. In: KDD 2002. pp. 550–556 (2002)
- [11] Ukkonen, E.: On-line construction of suffix trees. *Algorithmica* 14(3), 249–260 (1995)
- [12] Wang, Y., Zheng, Y., Xue, Y.: Travel time estimation of a path using sparse trajectories. In: KDD 2014. pp. 25–34 (2014)
- [13] Weiner, P.: Linear pattern-matching algorithms. In: Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory. pp. 1–11 (1973)
- [14] Westbrook, J.: Fast incremental planarity testing. In: ICALP 1992. pp. 342–353 (1992)

## A Appendix (Figures)

In this appendix, we show some supplemental figures which support understanding of the contents in the main body of this paper.

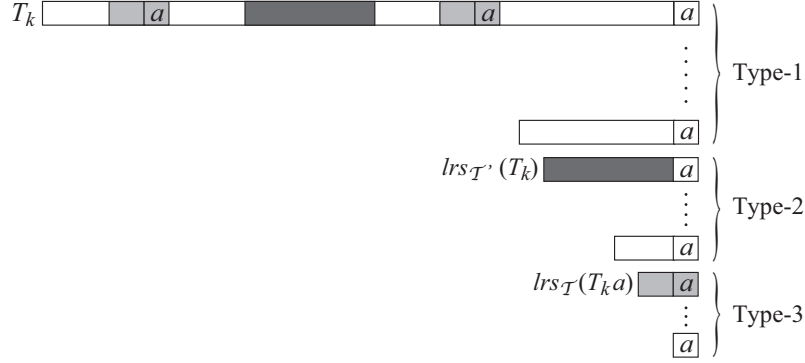


Figure 2: Illustration for the Type-1, Type-2, and Type-3 suffixes of  $T_k a$ .

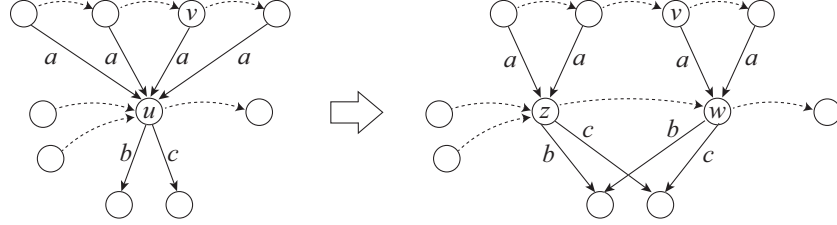


Figure 3: Illustration for node split in updating  $DAWG(\mathcal{T}')$  to  $DAWG(\mathcal{T})$ . The edge labeled with  $a$  from  $v$  to  $u = [lrs_{\mathcal{T}'}(T_k a)]_{\mathcal{T}'}$  is secondary, and hence  $u$  in  $DAWG(\mathcal{T}')$  is split into two nodes  $z = [Xa]_{\mathcal{T}}$  and  $w = [lrs_{\mathcal{T}}(T_k a)]_{\mathcal{T}}$  in  $DAWG(\mathcal{T})$ . The out-going edges of  $u$  are copied for  $w$ . The suffix links that point to  $u$  in  $DAWG(\mathcal{T}')$  point to  $z$  in  $DAWG(\mathcal{T})$ , and the suffix link from  $u$  in  $DAWG(\mathcal{T}')$  is from  $w$  in  $DAWG(\mathcal{T})$ . The suffix link from  $z$  is set to  $w$ . The time cost required for this node split is linear in the number of new nodes, edges, and suffix links.

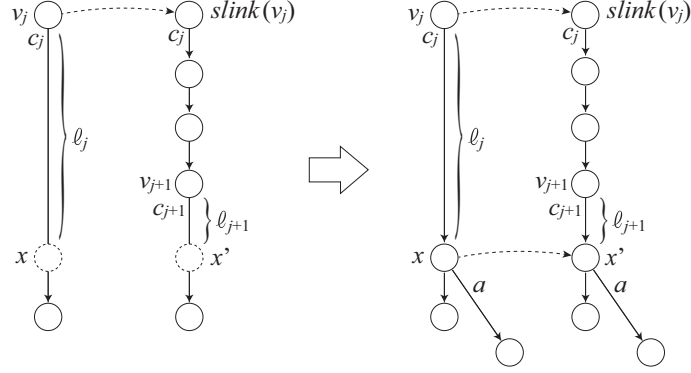


Figure 4: Illustration for Case II-b(ii), where the suffix tree is being updated by an update operator  $(k, a)$ . To the left is a part of the suffix tree just before the leaf corresponding to the  $j$ th suffix of  $T_k a$  is going to be inserted from the active point  $x$ . Since there is yet no suffix link from the locus for  $x$ , we move to the next active point  $x'$  via  $slink(v_j)$ , going down along the corresponding path from  $slink(v_j)$  to  $v_{j+1}$ . To the right is a part of the suffix tree after the leaves corresponding to the  $j$ th and  $(j + 1)$ th suffixes of  $T_k a$  have been inserted. The amount of work here is  $O((\ell_j - \ell_{j+1} + 1) \log \sigma)$ .

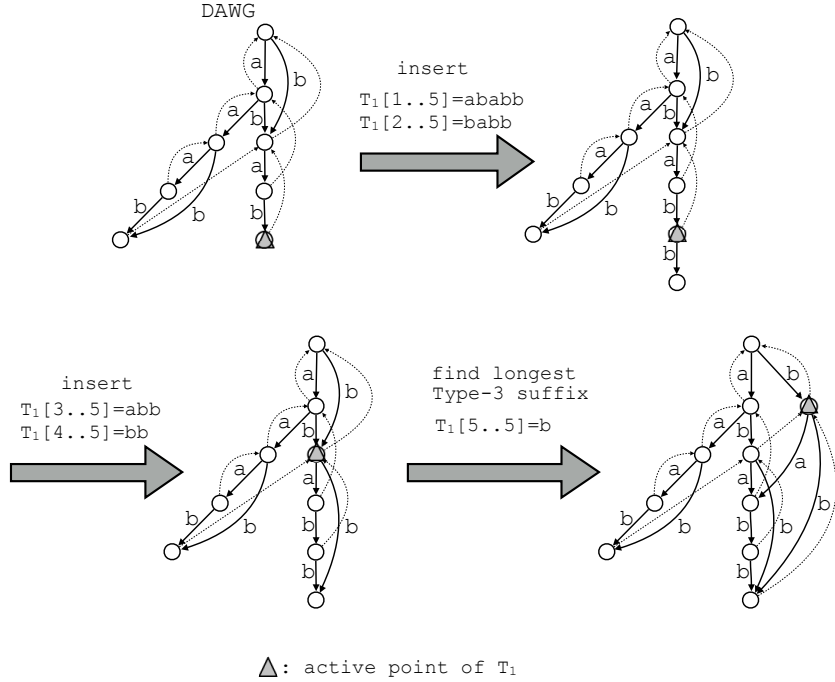


Figure 5: A snapshot of fully-online DAWG construction, where we update  $DAWG(\mathcal{T}')$  to  $DAWG(\mathcal{T})$  with  $\mathcal{T}' = \{T_1 = \text{abab}, T_2 = \text{aaab}\}$  and  $\mathcal{T} = \{T_1 b, T_2\}$ . We insert the suffixes of  $T_1 b$  as follows. Type-1 suffixes **ababb** and **babbb** are inserted by a new edge labeled **b** from the active point to the new sink. The active point moves via the suffix link, and Type-2 suffixes **abb** and **bb** are inserted by another new edge **b** from the active point to the new sink. The active point moves via the suffix link again, and the longest Type-3 suffix **b** is found. Since the edge from the source to the node is secondary, the node is separated into two nodes.

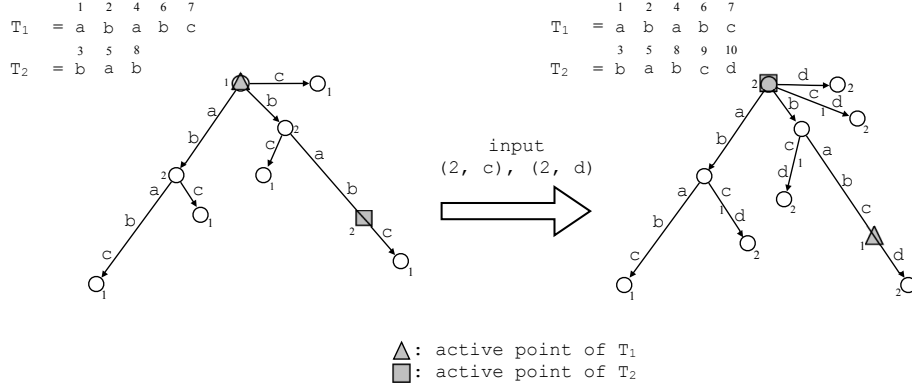


Figure 6: Consider a text collection  $\mathcal{T}$  with two texts which grow according to fully-online sequence  $U = (1, a), (1, b), (2, b), (1, a), (2, a), (1, b), (1, c), (2, b), (2, c), (2, d)$  of 10 update operators. To the left is  $STree(\mathcal{T}_{U[1..8]})$ , where the active point for  $T_1 = ababc$  is on the root and that for  $T_2 = bab$  is on the implicit node  $bab$ . The numbers 1 and 2 shown on  $STree(\mathcal{T}_{U[1..8]})$  indicate the loci of the suffixes of  $T_1$  and  $T_2$ , respectively. In  $STree(\mathcal{T}_{U[1..8]})$ , the labels of the in-coming edges to the leaves corresponding to  $babc$ ,  $abc$ , and  $bc$  are represented by triples  $\langle 1, 3, \infty \rangle$ ,  $\langle 1, 5, \infty \rangle$ , and  $\langle 1, 5, \infty \rangle$ , respectively. To the right is  $STree(\mathcal{T}_{U[1..10]})$ , where the 2nd text  $T_2$  has been updated from  $bab$  to  $babcd$ . Due to this update to  $T_2$ , the locus of the active point of  $T_1 = ababc$  has been changed to the implicit node  $babc$  (Difficulty A). Moreover, due to this update to  $T_2$ , the leaves representing  $babc$ ,  $abc$ , and  $bc$  have been respectively extended to representing  $babcd$ ,  $abcd$ , and  $bcd$ . Hence, the triples for the labels of their in-coming edges have to be updated to  $\langle 2, 2, \infty \rangle$ ,  $\langle 2, 4, \infty \rangle$ , and  $\langle 2, 4, \infty \rangle$ , respectively (Difficulty C).

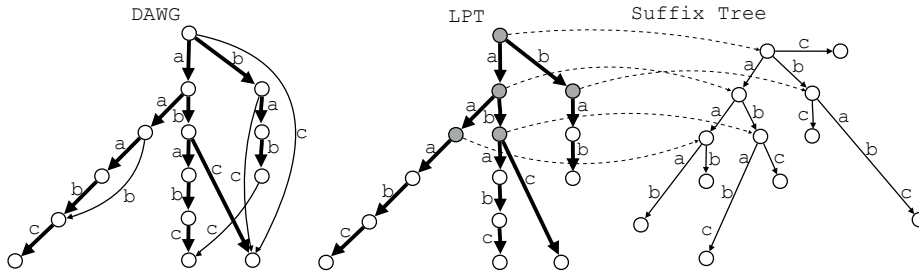


Figure 7: Illustration for  $DAWG(\mathcal{T})$ ,  $LPT(\mathcal{T})$ , and  $STree(\mathcal{T}')$ , where  $\mathcal{T}' = \{T_1 = aaab, T_2 = ababc, T_3 = ab\}$  and  $\mathcal{T} = \{T_1, T_2, T_3\}$ . The bold solid arrows represent the primary edges of  $DAWG(\mathcal{T})$ , the gray nodes are the marked nodes of  $LPT(\mathcal{T})$ , and the dashed arrows represent the links between the marked nodes of  $LPT(\mathcal{T})$  and the corresponding branching nodes of  $STree(\mathcal{T}')$ .  $lrs_{\mathcal{T}}(T_1c) = abc$ , and hence we perform an NMA query from node  $abc$  on  $LPT(\mathcal{T})$ , obtaining node  $ab$ . We then access the suffix tree node  $ab$  using the pointer from  $LPT(\mathcal{T})$ , and obtain the locus of  $abc$  on  $STree(\mathcal{T}')$ .

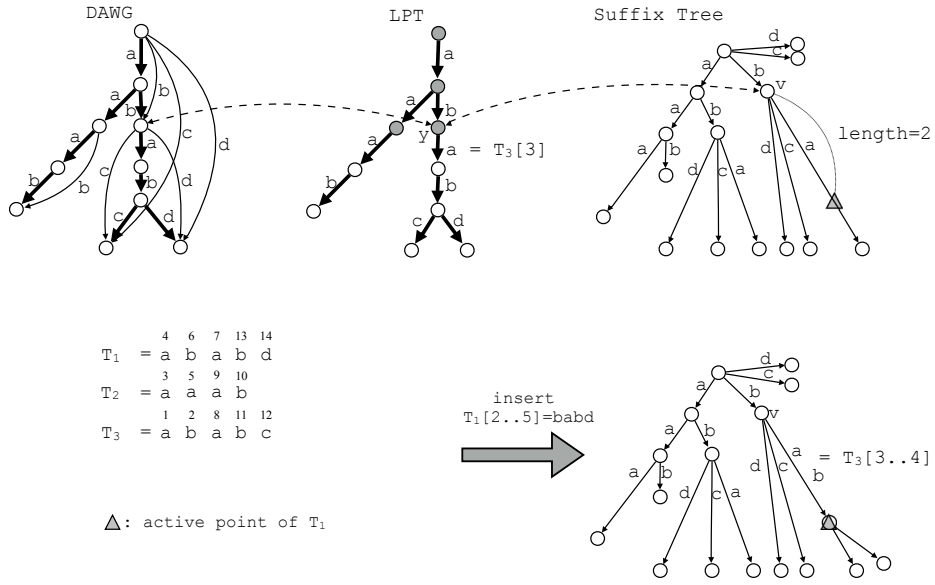


Figure 8: Illustration of how to determine the label of the in-coming edge of a new internal explicit node which is created on an edge leading to an existing leaf. Let  $\mathcal{T}' = \{T_1 = \text{abab}, T_2 = \text{aaab}, T_3 = \text{ababc}\}$ , and  $\mathcal{T} = \{T_1\text{d}, T_2, T_3\}$ . Now we are inserting a new leaf w.r.t. Type-2 suffix **babd** of  $T_1\text{d}$ . The canonical reference to the insertion point of this suffix is  $(\mathbf{b}, \mathbf{a}, 2)$ , and hence we create a new internal node on the middle of the out-going edge of node **b** whose edge label begins with **a**. Now, since  $\text{long}([b]\mathcal{T}) = \text{ab}$ , we access the LPT node  $y = \text{ab}$ . Since the label **a** of the out-going edge of  $y$  in  $LPT(\mathcal{T})$  is now represented by pair  $\langle 3, 3 \rangle$ , we can label the new suffix tree edge leading to the new internal node by  $\langle 3, 3, 3+2-1 \rangle = \langle 3, 3, 4 \rangle$ .

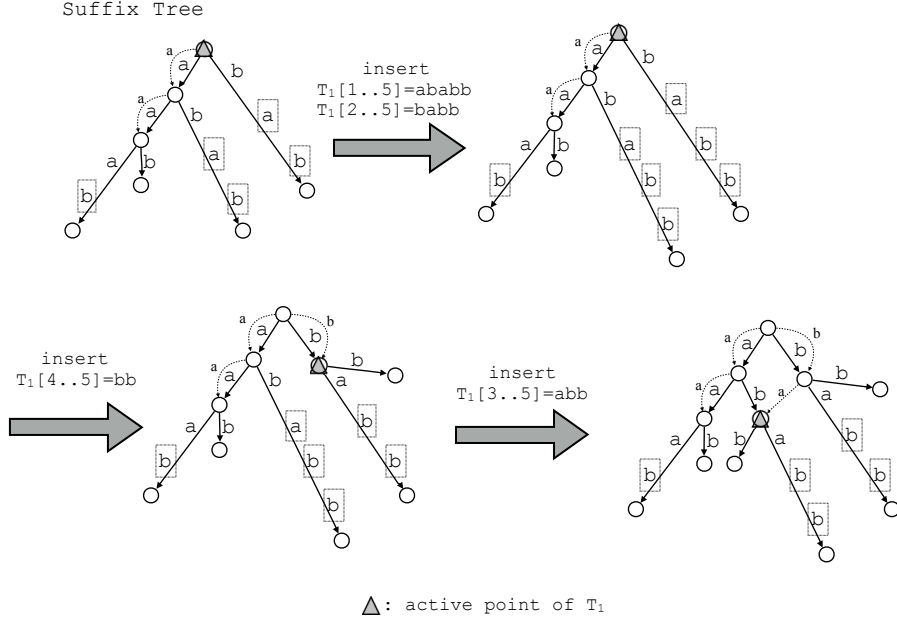


Figure 9: A snapshot of fully-online suffix tree construction, where we update  $STree(\mathcal{T}')$  to  $STree(\mathcal{T})$  with  $\mathcal{T}' = \{T_1 = \text{abab}, T_2 = \text{aaab}\}$  and  $\mathcal{T} = \{T_1\mathbf{b}, T_2\}$ . Recall that we employ lazy maintenance of the leaves, and hence each character within a box is only imaginary and is not computed during the updates. Due to lazy representation of leaves, we do nothing to insert the Type-1 suffixes of  $T_1\mathbf{b}$ . To start inserting the Type-2 suffixes in increasing order of length, we first find the longest Type-3 suffix  $\mathbf{b}$  via  $LPT(\mathcal{T})$  using Lemma 1. We insert the shortest Type-2 suffix  $\mathbf{bb}$ . Using Lemma 3 and Remark 1 in Appendix B, we find the edge whose label begins with  $\mathbf{b}$  from the root, and create a new internal node in the middle of this edge. After creating a new leaf from the new internal node and its in-coming edge with the first character label  $\mathbf{b}$ , we determine the label of the in-coming edge of the new internal node using Lemma 2. The reversed suffix link is set from the root to this new internal node  $\mathbf{b}$ . The next Type-2 suffix is  $\mathbf{abb}$ , and hence we query  $(v, \mathbf{a})$  to our suffix oracle of Lemma 2, where  $v$  is the node representing  $\mathbf{b}$ , and obtain node  $\mathbf{a}$ . We find the edge whose label begins with  $\mathbf{b}$  from this node, and create a new internal node in the middle of this edge. After creating a new leaf from the new internal node and its in-coming edge with the first character label  $\mathbf{b}$ , we determine the label of the in-coming edge of the new internal node using Lemma 2. The reversed suffix link is set from node  $\mathbf{b}$  to this new internal node  $\mathbf{ab}$ . Since we have inserted all the Type-2 suffixes, the update finishes.

## B Appendix (Proof of Lemma 2)

In this appendix, we show a complete proof of Lemma 2.

We use the following known result in our proof:

**Lemma 4** (Lowest common ancestor (LCA) on semi-dynamic tree [7]). *A semi-dynamic rooted tree can be maintained in linear space in its size so that the following operations are supported in worst-case  $O(1)$  time: 1) find the lowest common ancestor (LCA) of any two nodes; 2) insert a new node.*

We are ready to show Lemma 2.

*Proof.* The design of our suffix tree oracle follows the data structure by Fischer and Gawrychowski [9], but ours is much simpler since an amortized  $O(\log \sigma)$ -bound is enough for our goal. We define the *weight* of each node  $v$  of the suffix tree, denoted  $w(v)$ , to be the sum of the number of leaves in the subtree rooted at  $v$  and the number of reversed suffix links defined in the subtree. A node  $v$  is called *heavy* if  $w(v) \geq 2\sigma$ , and is called *light* if  $w(v) \leq \sigma$ . A node  $v$  with  $\sigma < w(v) < 2\sigma$  can be either light or heavy. Clearly, if a node is heavy, then its all ancestors are heavy. A heavy node  $v$  is called a *heavy leaf* if no children of  $v$  are heavy, and it is called a *heavy branching node* if at least two children of  $v$  are heavy. See also Fig. 10.

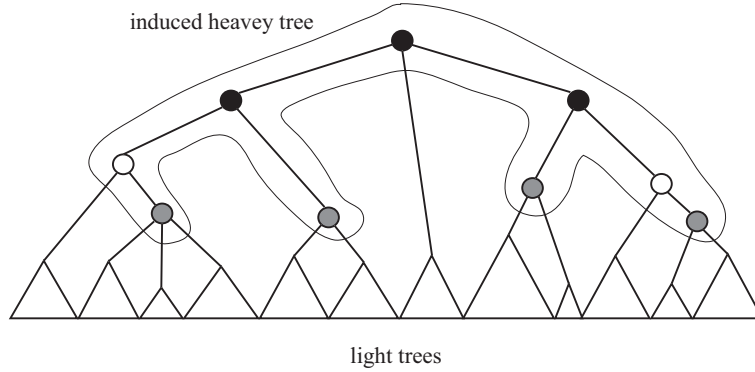


Figure 10: Illustration for heavy nodes, light trees, and induced heavy tree on a suffix tree. The circles represent heavy nodes, while the white triangles represent light trees. The gray nodes are heavy leaves, and the black nodes are branching heavy nodes. The induced heavy tree is a tree consisting only of these black and gray nodes.

First, we show a suffix tree oracle for heavy nodes. We maintain a tree called the *induced heavy tree* over the suffix tree which consists only of the heavy leaves and the heavy branching nodes. Since there are only  $O(n/\sigma)$  heavy leaves, the total size of the induced heavy tree is  $O(n/\sigma)$ . From each heavy node of the suffix tree, we maintain a pointer to its corresponding edge in the induced heavy tree. For each edge  $e$  of the induced heavy tree, if there is a suffix tree node  $v$  associated to  $e$  with  $rslink_b(v)$  defined for character  $b \in \Sigma$ , then we maintain an invariant  $lowest_e(b)$  which indicates the lowest node associated to  $e$  for which  $rslink_b(v)$  is defined. In each edge, we maintain these invariants for all characters by a balanced binary search tree. Since the size of a balanced binary search tree is  $O(\sigma)$ , the total space for all edges of the induced heavy tree is  $O(\sigma \times n/\sigma) = O(n)$ . We also maintain  $\sigma$  NMA data structures of Westbrook [14] over the induced heavy tree: A node  $u$  in the induced heavy tree is marked in the NMA data structure for character  $b \in \Sigma$ , iff  $lowest_e(b)$  is defined where  $e$  is the in-coming edge to  $u$ . Note that the total size for all  $\sigma$  NMA data structures is  $O(\sigma \times n/\sigma) = O(n)$  as well. Given a query  $(v, b)$  to the suffix tree oracle where  $v$  is a heavy node of the suffix tree, then we first access the edge  $e$  of the induced heavy tree with which  $v$  is associated. There are three cases:

- (1) If  $lowest_e(b)$  is an ancestor of  $v$ , then  $lowest_e(b)$  is the answer.
- (2) If  $lowest_e(b)$  is a descendant of  $v$ , then  $v$  is the answer.
- (3) If  $lowest_e(b)$  is not defined, then we take the branching node  $u$  of the induced heavy tree of which  $e$  is an out-going edge. We perform an NMA query from  $u$  using the NMA data structure associated with  $b$ , and then this case reduces to either case (1) or case (2).

This suffix tree oracle answers a query in amortized  $O(\log \sigma)$  time, since we need amortized  $O(1)$  time for each NMA query, and  $O(\log \sigma)$  time to search for  $lowest_b(e)$  in the balanced search tree and to access the NMA data structure for character  $b$ .

Second, we show a suffix tree oracle for light nodes. Each maximal subtree consisting only of light nodes is called a *light tree*. Clearly, the total number of nodes and reversed suffix links defined in each light tree is at most  $2\sigma - 1$ . For each light tree, we maintain a simple suffix tree oracle proposed by Fischer and Gawrychowski [9] which answers queries in  $O(\log \sigma)$  time: Consider any light tree  $LT$ . For each character  $b$  we maintain a preorder traversal of  $LT$  which contains all and only the nodes  $x$  in  $LT$  such that  $rslink_b(x)$  is defined. Then, for any query  $(v, b)$ ,  $u$  is the nearest marked ancestor of  $v$  with  $rslink_b(u)$  defined, iff  $v$  is the predecessor of  $u$  in the preorder traversal for character  $b$ . Since comparing two elements there reduces to computing their LCA, we can use the dynamic LCA data structure of Lemma 4. Thus, by maintaining a balanced search tree which stores the preorder traversal of  $LT$  for each character  $b$ , we can compute the predecessor in  $O(\log \sigma)$  time. The total size of the balanced search trees for all characters is linear in the number of reversed suffix links defined in  $LT$ , which is  $O(\sigma)$ .

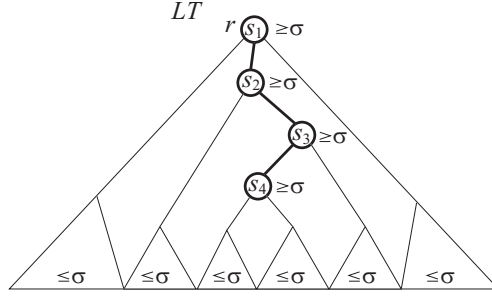


Figure 11: As soon as the weight of the root  $r$  of a light tree  $LT$  reaches  $2\sigma$ , we take a maximal path  $s_1, \dots, s_h$  of heavy nodes of weight at least  $\sigma$  from  $r$ . All these nodes  $s_1, \dots, s_h$  are then associated to an edge of the induced heavy tree. Since the weight of  $r$  is  $2\sigma$ , all the new light trees are of weight at most  $\sigma$ .

What remains is how to update the suffix tree oracle when a new leaf or a new reversed suffix link is inserted to the suffix tree. Assume that a new leaf or a new reversed suffix link is inserted to a light tree  $LT$  of size  $2\sigma - 1$ . The weight  $w(r)$  of the root  $r$  of  $LT$  is now  $2\sigma$ , meaning that the root  $r$  becomes a heavy node. We take a maximal path  $s_1, \dots, s_h$  starting from the root  $r = s_1$  such that  $w(s_i) \geq \sigma$  for all  $1 \leq i \leq h$  (see also Fig. 11 for illustration). We create pointers from these nodes to an edge of the induced heavy tree, so  $s_h$  becomes a new heavy leaf. Let  $p(s_1)$  be the parent of  $s_1$  in the original suffix tree. We update the induced heavy tree as follows.

- (a) If  $p(s_1)$  is already a heavy branching node, then we create a new edge from  $p(s_1)$  to  $s_h$  in the induced heavy tree and make pointers from  $s_1, \dots, s_h$  to this edge.
- (b) If  $p(s_1)$  is a heavy leaf, then we create pointers from  $s_1, \dots, s_h$  to the in-coming edge of  $p(s_1)$  in the induced heavy tree. This “extends” the in-coming edge of the induced heavy tree to the new heavy leaf  $s_h$ .
- (c) If  $p(s_1)$  has just become a new heavy branching node because of the new heavy leaf  $s_h$ , then  $p(s_1)$  becomes a new internal node of the induced heavy tree. Let  $e$  be the original edge split by  $p(s_1)$ . Note that we need to update the pointers to  $e$ . To do this efficiently, we use the “take the smaller” strategy: If at most half of the pointers from the suffix tree nodes to  $e$  are associated to the upper split part of  $e$ , then we redirect the pointers to the upper part of  $e$  to a newly created edge  $e'$  which is now the upper split part of  $e$ . We shorten  $e$  by making its starting point to  $p(s_1)$ , which is now the lower split part. The cost of redirecting the pointers in the “smaller” part can be charged to the unredirected pointers which remain in  $e$  (the “larger” part), and hence the amortized cost for redirection per pointer is  $O(1)$ . The other case can be treated symmetrically. Finally, we create a new edge from  $p(s_1)$  to  $s_h$  in the induced heavy tree and make pointers from  $s_1, \dots, s_h$  to this new edge.

It takes amortized  $O(\log \sigma)$  time to update the NMA data structures and balanced search trees for  $lowest(\cdot)$  for the split edge and the new edge in the heavy induced tree. We update the light trees as follows. By taking the nodes  $s_1, \dots, s_h$  from the original light tree  $LT$ , a number of light trees are created. We reconstruct the suffix tree oracle for each of these light trees. This takes time linear in the size of each tree, and since the size of each tree is at most  $\sigma$ , this takes  $O(\sigma)$  time. We can charge this cost to the  $\sigma$  new leaves and reversed suffix links to be inserted to each light tree, which will make the root of this light tree a heavy one. Thus, the amortized cost for reconstructing the suffix tree oracle for the light trees is



$O(1)$ . Thus, it requires amortized  $O(\log \sigma)$  time to update our suffix tree oracle per insertion of a new leaf or a new suffix link.

In the above description, we have assumed that the alphabet size  $\sigma$  is known beforehand. If it is not the case, then we can reconstruct the suffix oracle each time the alphabet size doubles due to the growth of the texts in the collection. Since the number of distinct characters in the texts is at most the total length of the texts, the amortized cost of the reconstruction is  $O(\log \sigma)$ .  $\square$

**Remark 1.** Consider an update operator  $(k, a)$  to the text collection. Recall that we want to insert the Type-2 suffixes of  $T_k a$  into the suffix tree in increasing order of length. Let  $xa$  be either the shortest Type-3 suffix of  $T_k a$  or any Type-2 suffix of  $T_k a$ . Let  $v$  be the lowest branching ancestor of  $x$ , and let  $bxa$  be the next Type-2 suffix of  $T_k a$  to be inserted into the suffix tree, where  $b \in \Sigma$ . Our suffix tree oracle described above covers the case where  $\text{rslink}_b(u)$  is defined for some ancestor  $u$  of  $v$ , but does not cover the other case where  $\text{rslink}_b(u)$  is undefined for any ancestor  $u$  of  $v$ . However, in this case we can easily access the locus of  $bx$  in  $O(\log \sigma)$  time: First, we move to the root of the suffix tree, and then take its out-going edge of which label begins with  $b$ . By assumption, there is no explicit node in the path between the root and the implicit node  $bx$ , and hence we can obtain the locus for  $bx$  with a simple arithmetic.